SPRAL_SCALING
Sparse Matrix Scalings
C User Guide

This package generates various scalings (and matchings) of real sparse matrices.

Given a **symmetric** matrix A, it finds a diagonal matrix D such that the scaled matrix

$$\hat{A} = DAD$$

has specific numerical properties.

Given an **unsymmetric** or **rectangular** matrix A, it finds diagonal matrices D_r and D_c such that the scaled matrix

$$\hat{A} = D_r A D_c$$

has specific numerical properties.

The specific numerical properties delivered depends on the algorithm used:

Matching-based algorithms scale A such that the maximum (absolute) value in each row and column of $\hat{A}$ is exactly 1.0, where the entries of maximum value form a maximum cardinality matching. The **Hungarian algorithm** delivers an optimal matching slowly, whereas the **auction algorithm** delivers an approximate matching quickly.

Norm-equilibration algorithms scale A such that the infinity norm of each row and column of $\hat{A}$ is $1.0 \pm \tau$ (for some user specified tolerance τ).

Jonathan Hogg (STFC Rutherford Appleton Laboratory)

Major version history

2014-12-17 Version 1.0.0 Initial public release

1 Installation

Please see the SPRAL install documentation.

2 Usage overview

2.1 Calling sequences

Access to the package requires inclusion of either `spral.h` (for the entire `SPRAL` library) or `spral_scaling.h` (for just the relevant routines). i.e.

```
#include "spral.h"
```

Figure 1: Data format example matrix (symmetric)

$$\begin{pmatrix} 1.1 & 2.2 & & 3.3 & \\ 2.2 & & 4.4 & & \\ & 4.4 & 5.5 & & 6.6 \\ 3.3 & & & 7.7 & 8.8 \\ & & 6.6 & 8.8 & 9.9 \end{pmatrix}$$

The following functions are available to the user:

- `spral_scaling_default_auction_options()` initializes the options structure for the auction algorithm.
- `spral_scaling_default_equilib_options()` initializes the options structure for the norm equilibriation algorithm.
- `spral_scaling_default_hungarian_options()` initializes the options structure for the Hungarian algorithm.
- `spral_scaling_auction_sym()` and `spral_scaling_auction_unsym()` generate approximate matching-based scalings for symmetric and unsymmetric/rectangular matrices respectively using an auction algorithm.
- `spral_scaling_equilib_sym()` and `spral_scaling_equilib_unsym()` generate norm-equilibration scalings for symmetric and unsymmetric/rectangular matrices respectively.
- `spral_scaling_hungarian_sym()` and `spral_scaling_hungarian_unsym()` generate matching-based scalings for a symmetric and unsymmetric/rectangular matrices respectively using the Hungarian algorithm.

2.2 Data formats

2.2.1 Compressed Sparse Column (CSC) Format

This standard data format consists of the following data:

```
int    m;                 /* number of rows (unsymmetric matrix) */
int    n;                 /* number of columns */
int    ptr[ n+1 ];        /* column pointers */
int    row[ ptr[n]-1 ];   /* row indices */
double val[ ptr[n]-1 ];   /* numerical values */
```

Non-zero matrix entries are ordered by increasing column index and stored in the arrays `row[]` and `val[]` such that `row[k]` holds the row number and `val[k]` holds the value of the `k`-th entry. The `ptr[]` array stores column pointers such that `ptr[i]` is the position in `row[]` and `val[]` of the first entry in the `i`-th column, and `ptr[n]` is one more than the total number of entries. There must be no duplicate or out of range entries. Entries that are zero, including those on the diagonal, need not be specified.

For **symmetric matrices**, only the lower triangular entries of A should be supplied. For **unsymmetric matrices**, all entries in the matrix should be supplied.

Note that these routines offer **no checking** of user data, and the behaviour of these routines with misformatted data is undefined.

To illustrate the CSC format, the following arrays describe the symmetric matrix shown in Figure 1.

```
int n = 5;
int ptr[]    = { 0,             3,   4,        6,        8,    9 };
int row[]    = { 0,   1,   3,   2,   2,   4,   3,   4,   4 };
double val[] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };
```

3 Auction Algorithm

3.1 spral_scaling_auction_default_options()

To initialize a variable of type `struct spral_scaling_auction_options` the following routine is provided.

```
void spral_scaling_auction_default_options(struct spral_scaling_auction_options *options);
```

`*options` is the instance to be initialized.

3.2 spral_scaling_auction_sym()

To generate a scaling for a symmetric matrix using an auction algorithm such that the entry of maximum absolute value in each row and column is approximately 1.0,

```
void spral_scaling_auction_sym(int n, const int *ptr, const int *row, const double *val,
   double *scaling, int *match, const struct spral_scaling_auction_options *options,
   struct spral_scaling_auction_inform *inform);
```

`n`, `ptr[n+1]`, `row[ptr[n]]`, `val[ptr[n]]` must hold the lower triangular part of A in compressed sparse column format as described in Section 2.2.

`scaling[n]` holds, on exit, the diagonal of D. `scaling[i]` holds d_i, the scaling corresponding to the `i`-th row and column.

`match[m]` may be `NULL`. If it is non-`NULL`, then on exit it specifies the matching of rows to columns. Row `i` is matched to column `match[i]`, or is unmatched if `match[i]=-1`.

`*options` specifies the algorithmic options used by the subroutine, as explained in Section 3.4.

`*inform` is used to return information about the execution of the subroutine, as explained in Section 3.5.

3.3 spral_scaling_auction_unsym()

To generate a scaling for an unsymmetric or rectangular matrix using an auction algorithm such that the entry of maximum absolute value in each row and column is approximately 1.0,

```
void spral_scaling_auction_unsym(int m, int n, const int *ptr, const int *row,
   const double *val, double *rscaling, double *cscaling, int *match,
   const struct spral_scaling_auction_options *options,
   struct spral_scaling_auction_inform *inform);
```

`m`, `n`, `ptr[n+1]`, `row[ptr[n]]`, `val[ptr[n]]` must hold the lower triangular part of A in compressed sparse column format as described in Section 2.2.

`rscaling[m]` holds, on exit, the diagonal of D_r. `rscaling[i]` holds d_i^r, the scaling corresponding to the `i`-th row.

`cscaling[n]` holds, on exit, the diagonal of D_c. `cscaling[j]` holds d_j^c, the scaling corresponding to the `j`-th column.

`match[m]` may be `NULL`. If it is non-`NULL`, then on exit it specifies the matching of rows to columns. Row `i` is matched to column `match[i]`, or is unmatched if `match[i]=-1`.

`*options` specifies the algorithmic options used by the subroutine, as explained in Section 3.4.

`*inform` is used to return information about the execution of the subroutine, as explained in Section 3.5.

3.4 struct spral_scaling_auction_options

The structure spral_scaling_auction_options is used to specify the options used by the routines spral_scaling_auction_sym() and spral_scaling_auction_unsym(). The components, that must be given default values through a call to spral_scaling_default_auction_options(), are:

int array_base specifies the array indexing base. It must have the value either 0 (C indexing) or 1 (Fortran indexing). If array_base=1, the entries of arrays ptr[],row[] and match[] start at 1, not 0. Further, entries of match[] that are unmatched are indicated by a value of 0, not -1. The default value is array_base=0.

float eps_initial specifies the initial value of the minimum improvement parameter ϵ as described in Section 3.7.

int max_iterations specifies the maximum number of iterations the algorithm may perform. The default is max_iterations=30000.

int max_unchanged[3] specifies, together with min_proportion[], the termination conditions for the algorithm, as described in Section 3.7. The default is max_unchanged[] = { 10, 100, 100 }.

float min_proportion[3] specifies, together with max_unchanged(:), the termination conditions for the algorithm, as described in Section 3.7. The default is min_proportion[] = { 0.90, 0.0, 0.0 }.

3.5 struct spral_scaling_auction_inform

The structure spral_scaling_auction_inform is used to hold parameters that give information about the progress of the routines spral_scaling_auction_sym() and spral_scaling_auction_unsym(). The components are:

int flag gives the exit status of the algorithm (details in Section 3.6).

int iterations holds the number of iterations performed.

int matched holds the number of rows and columns that have been matched. As the algorithm may terminate before a full matching is obtained, this only provides a lower bound on the structural rank.

int stat holds, in the event of an allocation error, the Fortran stat parameter if it is available (and is set to 0 otherwise).

int unmatchable holds the number of columns designated as unmatchable. A column is designated as unmatchable if there is no way to match it that improves the quality of the matching. It provides an approximate lower bound on the structural rank deficiency.

3.6 Error Flags

A successful return from a routine is indicated by inform.flag having the value zero. A negative value is associated with an error message.

Possible negative (error) values are:

-1 Allocation error. If available, the Fortran stat parameter is returned in inform.stat.

3.7 Algorithm description

This algorithm finds a fast approximation to the matching and scaling produced by the HSL package MC64. If an optimal matching is required, use the Hungarian algorithm instead. The algorithm works by solving the following maximum product optimization problem using an auction algorithm. The scaling is derived from the dual variables associated with the solution.

$$\begin{aligned} \max_{\sigma} \quad & \prod_{i=1}^{m}\prod_{j=1}^{n} |a_{ij}|\sigma_{ij} \\ \text{s.t.} \quad & \sum_{i=1}^{m} \sigma_{ij} = 1, \qquad \forall j = 1, n \\ & \sum_{j=1}^{n} \sigma_{ij} = 1, \qquad \forall i = 1, m \\ & \sigma_{ij} \in \{0, 1\}. \end{aligned}$$

The array σ gives a matching of rows to columns.

By using the transformation

$$w_{ij} = \log c_j - \log |a_{ij}|,$$

where $c_j = \max_i |a_{ij}|$, the maximum product problem in a_{ij} is replaced by a minimum sum problem in w_{ij} where all entries are positive. By standard optimization theory, there exist dual variables u and v corresponding to the constraints that satisfy the first order optimality conditions

$$\begin{aligned} w_{ij} - u_i - v_j &= 0, \qquad \text{if } \sigma_{ij} = 1, \\ w_{ij} - u_i - v_j &\geq 0, \qquad \text{if } \sigma_{ij} = 0. \end{aligned}$$

To obtain a scaling we define scaling matrices D_r and D_c as

$$\begin{aligned} d_i^r &= e^{u_i}, \\ d_i^c &= e^{v_i}. \end{aligned}$$

If a symmetric scaling is required, we average these as

$$d_i = \sqrt{d_i^r d_i^c}.$$

By the first order optimality conditions, these scaling matrices guarantee that

$$\begin{aligned} d_i^r |a_{ij}| d_j^c &= 1, \qquad \text{if } \sigma_{ij} = 1, \\ d_i^r |a_{ij}| d_j^c &\leq 1, \qquad \text{if } \sigma_{ij} = 0. \end{aligned}$$

To solve the minimum sum problem an auction algorithm is used. The algorithm is *not* guaranteed to find an optimal matching. However it can find an approximate matching very quickly. A matching is maintained along with the row pricing vector u. In each major iteration, we loop over each column in turn. If the column j is unmatched, we calculate the value $p_i = w_{ij} - u_i$ for each entry and find the maximum across the column. If this maximum is positive, the current matching can be improved by matching column j with row i. This may mean that the previous match of row i now becomes unmatched. We update the price of row i, that is u_i, to reflect this new benefit and continue to the next column.

To prevent incremental shuffling, we insist that the value of adding a new column is at least a threshold value ϵ above zero, where ϵ is based on the last iteration in which row i changed its match. This is done by adding ϵ to the price u_i, where $\epsilon =$ `options.eps_initial` $+$ `itr`$/(n+1)$, where `itr` is the current iteration number.

The algorithm terminates if any of the following are satisfied:

- All entries are matched.
- The number of major iterations exceeds `options.max_iterations`.
- At least `options.max_unchanged[0]` iterations have passed without the cardinality of the matching increasing, and the proportion of matched columns is `options.min_proportion[0]`.
- At least `options.max_unchanged[1]` iterations have passed without the cardinality of the matching increasing, and the proportion of matched columns is `options.min_proportion[1]`.
- At least `options.max_unchanged[2]` iterations have passed without the cardinality of the matching increasing, and the proportion of matched columns is `options.min_proportion[2]`.

The different combinations given by `options.max_unchanged[0:2]` and `options.min_proportion[0:2]` allow a wide range of termination heuristics to be specified by the user depending on their particular needs. Note that the matching and scaling produced will always be approximate as ϵ is non-zero.

Further details are given in the following paper:

[1] J.D. Hogg and J.A. Scott. (2014). On the efficient scaling of sparse symmetric matrices using an auction algorithm. RAL Technical Report RAL-P-2014-002.

3.8 Example of spral_scaling_auction_sym()

The following code shows an example usage of spral_scaling_auction_sym().

```
/* examples/C/scaling/auction_sym.f90 - Example code for SPRAL_SCALING */
#include <stdlib.h>
#include <stdio.h>
#include "spral.h"

void main(void) {
   /* Derived types */
   struct spral_scaling_auction_options options;
   struct spral_scaling_auction_inform inform;

   /* Other variables */
   int match[5], i, j;
   double scaling[5];

   /* Data for symmetric matrix:
    * ( 2  1         )
    * ( 1  4  1    8 )
    * (    1  3  2   )
    * (       2      )
    * (    8       2 ) */
   int n = 5;
   int ptr[]    = { 0,        2,             5,      7,7,   8 };
   int row[]    = { 0,   1,   1,   2,   4,   2,   3,   4   };
   double val[] = { 2.0, 1.0, 4.0, 1.0, 8.0, 3.0, 2.0, 2.0 };
   printf("Initial matrix:\n");
   spral_print_matrix(-1, SPRAL_MATRIX_REAL_SYM_INDEF, n, n, ptr, row, val, 0);

   /* Perform symmetric scaling */
   spral_scaling_auction_default_options(&options);
   spral_scaling_auction_sym(n, ptr, row, val, scaling, match, &options, &inform);
   if(inform.flag<0) {
      printf("spral_scaling_auction_sym() returned with error %5d", inform.flag);
      exit(1);
   }

   /* Print scaling and matching */
   printf("Matching:");
   for(int i=0; i<n; i++) printf(" %10d", match[i]);
   printf("\nScaling: ");
   for(int i=0; i<n; i++) printf(" %10.2le", scaling[i]);
   printf("\n");

   /* Calculate scaled matrix and print it */
   for(int i=0; i<n; i++) {
      for(int j=ptr[i]; j<ptr[i+1]; j++)
         val[j] = scaling[row[j]] * val[j] * scaling[i];
   }
   printf("Scaled matrix:\n");
   spral_print_matrix(-1, SPRAL_MATRIX_REAL_SYM_INDEF, n, n, ptr, row, val, 0);
}
```

The above code produces the following output.

```
Initial matrix:
Real symmetric indefinite matrix, dimension 5x5 with 8 entries.
0:   2.0000E+00   1.0000E+00
1:   1.0000E+00   4.0000E+00   1.0000E+00                8.0000E+00
2:                1.0000E+00   3.0000E+00   2.0000E+00
3:                             2.0000E+00
4:                8.0000E+00                             2.0000E+00
Matching:          0          4          3          2          1
Scaling:    7.07e-01   1.62e-01   2.78e-01   1.80e+00   7.72e-01
Scaled matrix:
Real symmetric indefinite matrix, dimension 5x5 with 8 entries.
0:   1.0000E+00   1.1443E-01
1:   1.1443E-01   1.0476E-01   4.5008E-02                1.0000E+00
2:                4.5008E-02   2.3204E-01   1.0000E+00
3:                             1.0000E+00
4:                1.0000E+00                             1.1932E+00
```

4 Norm-equilibration algorithm

4.1 spral_scaling_equilib_default_options()

To initialize a variable of type `struct spral_scaling_equilib_options` **the following routine is provided.**

```
void spral_scaling_equilib_default_options(struct spral_scaling_equilib_options *options);
```

`*options` is the instance to be initialized.

4.2 spral_scaling_equilib_sym()

To generate a scaling for a symmetric matrix using a norm equilibration algorithm such that the infinity norm of each row and column is equal to $1.0 \pm \tau$**,**

```
void spral_scaling_equilib_sym(int n, const int *ptr, const int *row, const double *val,
   double *scaling, const struct spral_scaling_equilib_options *options,
   struct spral_scaling_equilib_inform *inform);
```

`n`, `ptr[n+1]`, `row[ptr[n]]`, `val[ptr[n]]` must hold the lower triangular part of A in compressed sparse column format as described in Section 2.2.

`scaling[n]` holds, on exit, the diagonal of D. `scaling[i]` holds d_i, the scaling corresponding to the `i`-th row and column.

`*options` specifies the algorithmic options used by the subroutine, as explained in Section 4.4.

`*inform` is used to return information about the execution of the subroutine, as explained in Section 4.5.

4.3 spral_scaling_equilib_unsym()

To generate a scaling for an unsymmetric or rectangular matrix using a norm equilibration algorithm such that the infinity norm of each row and column is equal to $1.0 \pm \tau$**,**

```
void spral_scaling_equilib_unsym(int m, int n, const int *ptr, const int *row,
   const double *val, double *rscaling, double *cscaling,
   const struct spral_scaling_equilib_options *options,
   struct spral_scaling_equilib_inform *inform);
```

`m`, `n`, `ptr[n+1]`, `row[ptr[n]]`, `val[ptr[n]]` must hold the lower triangular part of A in compressed sparse column format as described in Section 2.2.

`rscaling[m]` holds, on exit, the diagonal of D_r. `scaling[i]` holds d_i^r, the scaling corresponding to the `i`-th row.

`cscaling[n]` holds, on exit, the diagonal of D_c. `scaling[j]` holds d_j^c, the scaling corresponding to the `j`-th column.

`*options` specifies the algorithmic options used by the subroutine, as explained in Section 4.4.

`*inform` is used to return information about the execution of the subroutine, as explained in Section 4.5.

4.4 struct spral_scaling_equilib_options

The structure `spral_scaling_equilib_options` is used to specify the options used by the routines `spral_scaling_equilib_sym()` and `spral_scaling_equilib_unsym()`. The components, that must be given default values through a call to `spral_scaling_default_equilib_options()`, are:

`int array_base` specifies the array indexing base. It must have the value either `0` (C indexing) or `1` (Fortran indexing). If `array_base=1`, the entries of arrays `ptr[]` and `row[]` start at 1, not 0. The default value is `array_base=0`.

`int max_iterations` specifies the maximum number of iterations the algorithm may perform. The default is `max_iterations=10`.

`float tol` specifies the convergence tolerance for the algorithm (though often termination is based on `max_iterations`). The default is `tol = 1e-8`.

4.5 struct spral_scaling_equilib_inform

The structure `spral_scaling_equilib_inform` is used to hold parameters that give information about the progress of the routines `spral_scaling_equilib_sym()` and `spral_scaling_equilib_unsym()`. The components are:

`int flag` gives the exit status of the algorithm (details in Section 4.6).

`int iterations` holds, on exit, the number of iterations performed.

`int stat` holds, in the event of an allocation error or deallocation error, the Fortran `stat` parameter if it is available (and is set to `0` otherwise).

4.6 Error Flags

A successful return from a routine is indicated by `inform.flag` having the value zero. A negative value is associated with an error message.

Possible negative (error) values are:

`-1` Allocation error. If available, the Fortran `stat` parameter is returned in `inform.stat`.

4.7 Algorithm description

This algorithm is very similar to that used by the HSL routine `MC77`. An iterative method is used to scale the infinity norm of both rows and columns to 1 with an asymptotic linear rate of convergence of $\frac{1}{2}$, preserving symmetry if the matrix is symmetric.

For unsymmetric matrices, the algorithm outline is as follows:

$D_r^{(0)} = I, D_c^{(0)} = I$

for $k = 1,$ `options.max_iterations` **do**

$A^{(k-1)} = D_r^{(k-1)} A D_c^{(k-1)}$

$(D_r^{(k)})_{ii} = (D_r^{(k-1)})_{ii} \;/\; \sqrt{\max_j (A^{(k-1)})_{ij}}$

$(D_c^{(k)})_{jj} = (D_c^{(k-1)})_{jj} \;/\; \sqrt{\max_i (A^{(k-1)})_{ij}}$

if$(|1 - \|A^{(k-1)}\|_{\max}| \leq$ `options.tol`**) exit**

end for

For symmetric matrices, $A^{(k-1)}$ is symmetric, so $D_r^{(k)} = D_c^{(k)}$, and some operations can be skipped.

Further details are given in the following paper:

[2] P. Knight, D. Ruiz and B. Ucar. (2012). A symmetry preserving algorithm for matrix scaling. INRIA Research Report 7552.

4.8 Example of spral_scaling_equilib_sym()

The following code shows an example usage of `spral_scaling_equilib_sym()`.

```
/* examples/C/scaling/equilib_sym.f90 - Example code for SPRAL_SCALING */
#include <stdlib.h>
#include <stdio.h>
#include "spral.h"

void main(void) {
   /* Derived types */
   struct spral_scaling_equilib_options options;
   struct spral_scaling_equilib_inform inform;

   /* Other variables */
   int i, j;
   double scaling[5];

   /* Data for symmetric matrix:
    * ( 2  1         )
    * ( 1  4  1    8 )
    * (    1  3  2   )
    * (       2      )
    * (    8       2 ) */
   int n = 5;
   int ptr[]    = { 0,        2,             5,      7,7,   8 };
   int row[]    = { 0,   1,   1,   2,   4,   2,   3,   4   };
   double val[] = { 2.0, 1.0, 4.0, 1.0, 8.0, 3.0, 2.0, 2.0 };
   printf("Initial matrix:\n");
   spral_print_matrix(-1, SPRAL_MATRIX_REAL_SYM_INDEF, n, n, ptr, row, val, 0);

   /* Perform symmetric scaling */
   spral_scaling_equilib_default_options(&options);
   spral_scaling_equilib_sym(n, ptr, row, val, scaling, &options, &inform);
   if(inform.flag<0) {
      printf("spral_scaling_equilib_sym() returned with error %5d", inform.flag);
      exit(1);
   }

   /* Print scaling */
   printf("Scaling: ");
   for(int i=0; i<n; i++) printf(" %10.2le", scaling[i]);
   printf("\n");

   /* Calculate scaled matrix and print it */
   for(int i=0; i<n; i++) {
      for(int j=ptr[i]; j<ptr[i+1]; j++)
         val[j] = scaling[row[j]] * val[j] * scaling[i];
   }
   printf("Scaled matrix:\n");
```

```
   spral_print_matrix(-1, SPRAL_MATRIX_REAL_SYM_INDEF, n, n, ptr, row, val, 0);
}
```

The above code produces the following output.

```
Initial matrix:
Real symmetric indefinite matrix, dimension 5x5 with 8 entries.
0:   2.0000E+00   1.0000E+00
1:   1.0000E+00   4.0000E+00   1.0000E+00                8.0000E+00
2:                1.0000E+00   3.0000E+00   2.0000E+00
3:                             2.0000E+00
4:                8.0000E+00                              2.0000E+00
Scaling:    7.07e-01   3.54e-01   5.77e-01   8.66e-01   3.54e-01
Scaled matrix:
Real symmetric indefinite matrix, dimension 5x5 with 8 entries.
0:   1.0000E+00   2.5000E-01
1:   2.5000E-01   5.0000E-01   2.0412E-01                1.0000E+00
2:                2.0412E-01   1.0000E+00   9.9960E-01
3:                             9.9960E-01
4:                1.0000E+00                              2.5000E-01
```

5 Hungarian algorithm

5.1 spral_scaling_hungarian_default_options()

To initialize a variable of type `struct spral_scaling_hungarian_options` the following routine is provided.

```
void spral_scaling_hungarian_default_options(struct spral_scaling_hungarian_options *options);
```

`*options` is the instance to be initialized.

5.2 spral_scaling_hungarian_sym()

To generate a scaling for a symmetric matrix using the Hungarian algorithm such that the entry of maximum absolute value in each row and column is 1.0,

```
void spral_scaling_hungarian_sym(int n, const int *ptr, const int *row, const double *val,
   double *scaling, int *match, const struct spral_scaling_hungarian_options *options,
   struct spral_scaling_hungarian_inform *inform);
```

`n`, `ptr[n+1]`, `row[ptr[n]]`, `val[ptr[n]]` must hold the lower triangular part of A in compressed sparse column format as described in Section 2.2.

`scaling[n]` holds, on exit, the diagonal of D. `scaling[i]` holds D_{ii}, the scaling corresponding to the `i`-th row and column.

`match[m]` may be `NULL`. If it is non-`NULL`, then on exit it specifies the matching of rows to columns. Row `i` is matched to column `match[i]`, or is unmatched if `match[i]=-1`.

`*options` specifies the algorithmic options used by the subroutine, as explained in Section 5.4.

`*inform` is used to return information about the execution of the subroutine, as explained in Section 5.5.

5.3 spral_scaling_hungarian_unsym()

To generate a scaling for an unsymmetric or rectangular matrix using the Hungarian algorithm such that the entry of maximum absolute value in each row and column is 1.0,

```
void spral_scaling_hungarian_unsym(int m, int n, const int *ptr, const int *row,
   const double *val, double *rscaling, double *cscaling, int *match,
   const struct spral_scaling_hungarian_options *options,
   struct spral_scaling_hungarian_inform *inform);
```

`m`, `n`, `ptr[n+1]`, `row[ptr[n]]`, `val[ptr[n]]` must hold the lower triangular part of A in compressed sparse column format as described in Section 2.2.

`rscaling[m]` holds, on exit, the diagonal of D_r. `rscaling[i]` holds d_i^r, the scaling corresponding to the `i`-th row.

`cscaling[n]` holds, on exit, the diagonal of D_c. `cscaling[j]` holds d_j^c, the scaling corresponding to the `j`-th column.

`match[m]` may be `NULL`. If it is non-`NULL`, then on exit it specifies the matching of rows to columns. Row `i` is matched to column `match[i]`, or is unmatched if `match[i]=-1`.

`*options` specifies the algorithmic options used by the subroutine, as explained in Section 5.4.

`*inform` is used to return information about the execution of the subroutine, as explained in Section 5.5.

5.4 struct spral_scaling_hungarian_options

The structure `spral_scaling_hungarian_options` is used to specify the options used by the routines `spral_scaling_hungarian_sym()` and `spral_scaling_hungarian_unsym()`. The components, that must be given default values through a call to `spral_scaling_default_hungarian_options()`, are:

`int array_base` specifies the array indexing base. It must have the value either `0` (C indexing) or `1` (Fortran indexing). If `array_base=1`, the entries of arrays `ptr[]`,`row[]` and `match[]` start at 1, not 0. Further, entries of `match[]` that are unmatched are indicated by a value of `0`, not `-1`. The default value is `array_base=0`.

`bool scale_if_singular` specifies whether scaling shuold continue if the matrix A is found to be structurally singular. If `scale_if_singular=true`, and the A is structurally singular, a partial scaling corresponding to a maximum cardinality matching will be returned and a warning issued. Otherwise an identity scaling will be returned and an error issued.

5.5 struct spral_scaling_hungarian_inform

The structure `spral_scaling_hungarian_inform` is used to hold parameters that give information about the progress of the routines `spral_scaling_hungarian_sym()` and `spral_scaling_hungarian_unsym()`. The components are:

`int flag` gives the exit status of the algorithm (details in Section 5.6).

`int matched` holds the number of rows and columns that have been matched (i.e. the structural rank).

`int stat` holds, in the event of an allocation error or deallocation error, the Fortran `stat` parameter if it is available (and is set to `0` otherwise).

5.6 Error Flags

A successful return from a routine is indicated by `inform.flag` having the value zero. A negative value is associated with an error message and a positive value with a warning.

Possible negative (error) values are:

`-1` Allocation error. If available, the Fortran `stat` parameter is returned in `inform.stat`.

`-2` Matrix A is structurally rank-deficient. This error is returned only if `options.scale_if_singular=false`. The scaling vector is set to `1.0` and a matching of maximum cardinality returned in the optional argument `match[]`, if present.

Possible positive (warning) values are:

`+1` Matrix A is structurally rank-deficient. This warning is returned only if `options.scale_if_singular=true`.

5.7 Algorithm description

This algorithm is the same as used by the HSL package MC64. A scaling is derived from dual variables found during the solution of the below maximum product optimization problem using the Hungarian algorithm.

$$\begin{array}{ll} \max\limits_{\sigma} & \prod_{i=1}^{m}\prod_{j=1}^{n}|a_{ij}|^{\sigma_{ij}} \\ \text{s.t.} & \sum_{i=1}^{m}\sigma_{ij}=1, \qquad \forall j=1,n \\ & \sum_{j=1}^{n}\sigma_{ij}=1, \qquad \forall i=1,m \\ & \sigma_{ij}\in\{0,1\}. \end{array}$$

The array σ gives a matching of rows to columns.

By using the transformation

$$w_{ij} = \log c_j - \log|a_{ij}|,$$

where $c_j = \max_i |a_{ij}|$, the maximum product problem in a_{ij} is replaced by a minimum sum problem in w_{ij} where all entries are positive. By standard optimization theory, there exist dual variables u and v corresponding to the constraints that satisfy the first order optimality conditions

$$\begin{aligned} w_{ij} - u_i - v_j &= 0, \qquad \text{if } \sigma_{ij}=1, \\ w_{ij} - u_i - v_j &\geq 0, \qquad \text{if } \sigma_{ij}=0. \end{aligned}$$

To obtain a scaling we define scaling matrices D_r and D_c as

$$\begin{aligned} d_i^r &= e^{u_i}, \\ d_i^c &= e^{v_i}. \end{aligned}$$

If a symmetric scaling is required, we average these as

$$d_i = \sqrt{d_i^r d_i^c}.$$

By the first order optimality conditions, these scaling matrices guarantee that

$$\begin{aligned} d_i^r|a_{ij}|d_j^c &= 1, \qquad \text{if } \sigma_{ij}=1, \\ d_i^r|a_{ij}|d_j^c &\leq 1, \qquad \text{if } \sigma_{ij}=0. \end{aligned}$$

To solve the minimum sum problem, the Hungarian algorithm maintains an optimal matching on a subset of the rows and columns. It proceeds to grow this set by finding augmenting paths from an unmatched row to an unmatched column. The algorithm is guaranteed to find the optimal solution in a fixed number of steps, but can be very slow as it may need to explore the full matrix a number of times equal to the dimension of the matrix. To minimize the solution time, a warmstarting heuristic is used to construct an initial optimal subset matching.
Further details are given in the following paper:

[3] I.S. Duff and J. Koster. (1997). The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. SIAM J. Matrix Anal. Applics. 20(4), pp 889–901.

5.8 Example usage of spral_scaling_hungarian_unsym()

The following code shows an example usage of `hungarian_scale_unsym()`.

```
/* examples/C/scaling/hungarian_unsym.f90 - Example code for SPRAL_SCALING */
#include <stdlib.h>
#include <stdio.h>
#include "spral.h"

void main(void) {
   /* Derived types */
   struct spral_scaling_hungarian_options options;
   struct spral_scaling_hungarian_inform inform;
```

```
   /* Other variables */
   int match[5], i, j;
   double rscaling[5], cscaling[5];

   /* Data for unsymmetric matrix:
    * ( 2  5         )
    * ( 1  4       7 )
    * (    1     2   )
    * (       3      )
    * (    8       2 ) */
   int m = 5, n = 5;
   int ptr[]    = { 0,        2,                  6,   7,   8,       10 };
   int row[]    = { 0,   1,   0,   1,   2,   4,   3,   2,   1,   4   };
   double val[] = { 2.0, 1.0, 5.0, 4.0, 1.0, 8.0, 3.0, 2.0, 7.0, 2.0 };
   printf("Initial matrix:\n");
   spral_print_matrix(-1, SPRAL_MATRIX_REAL_UNSYM, m, n, ptr, row, val, 0);

   /* Perform symmetric scaling */
   spral_scaling_hungarian_default_options(&options);
   spral_scaling_hungarian_unsym(m, n, ptr, row, val, rscaling, cscaling, match,
         &options, &inform);
   if(inform.flag<0) {
      printf("spral_scaling_hungarian_unsym() returned with error %5d", inform.flag);
      exit(1);
   }

   /* Print scaling and matching */
   printf("Matching:");
   for(int i=0; i<n; i++) printf(" %10d", match[i]);
   printf("\nRow Scaling: ");
   for(int i=0; i<m; i++) printf(" %10.2le", rscaling[i]);
   printf("\nCol Scaling: ");
   for(int i=0; i<n; i++) printf(" %10.2le", cscaling[i]);
   printf("\n");

   /* Calculate scaled matrix and print it */
   for(int i=0; i<n; i++) {
      for(int j=ptr[i]; j<ptr[i+1]; j++)
         val[j] = rscaling[row[j]] * val[j] * cscaling[i];
   }
   printf("Scaled matrix:\n");
   spral_print_matrix(-1, SPRAL_MATRIX_REAL_UNSYM, m, n, ptr, row, val, 0);
}
```

The above code produces the following output.

```
Initial matrix:
Real unsymmetric matrix, dimension 5x5 with * entries.
0:   2.0000E+00   5.0000E+00
1:   1.0000E+00   4.0000E+00                              7.0000E+00
2:                1.0000E+00                2.0000E+00
3:                              3.0000E+00
4:                8.0000E+00                              2.0000E+00
Matching:          0          4          3          2          1
Row Scaling:    5.22e-01   5.22e-01   5.22e-01   5.22e-01   5.22e-01
Col Scaling:    9.59e-01   2.40e-01   6.39e-01   9.59e-01   2.74e-01
Scaled matrix:
```

```
Real unsymmetric matrix, dimension 5x5 with * entries.
0:   1.0000E+00   6.2500E-01
1:   5.0000E-01   5.0000E-01                              1.0000E+00
2:                1.2500E-01                1.0000E+00
3:                              1.0000E+00
4:                1.0000E+00                              2.8571E-01
```